A Theory of Adaptable Contract-Based Service Composition

G. Bernardi M. Bugliesi D. Macedonio S. Rossi
Dipartimento di Informatica, Università Ca' Foscari, Venice, Italy
e-mail:{gbernard,bugliesi,mace,srossi}@dsi.unive.it

Abstract

Service Oriented Architectures draw heavily on techniques for reusing and assembling off-the-shelf software components. While powerful, this programming practice is not without a cost: the software architect must ensure that the off-the-shelf components interact safely and in ways that conform with the specification. We develop a new theory for adaptable service compositions. The theory provides an effective framework for analyzing the conformance of contract-based service compositions, and for enforcing their compliance, in a uniform, formally elegant setting.

1. Introduction

Modern software design is increasingly being based on a methodology that has become known as Service Oriented Architecture (SOA). Central to SOA is the idea of *reusing* existing off-the-shelf software units (services) and assembling them to develop new applications. Two approaches have emerged as mainstream: *orchestration* and *choreography*. An orchestration combines existing services around the orchestrator, a component that acts as the coordinator of the services and mediates all of their interactions. A choreography, instead, organizes the services along a message flow which all partners must comply with autonomously, without intervention of a central coordinator.

Several languages have recently emerged as specification formalisms for orchestrations (e.g., XLANG, WSFL and WS-BPEL) and choreographies (e.g., WS-CDL and BPEL4Chor). Such languages provide primitives for assembling services based on abstract descriptions of the services' behavior, given as interfaces or service *contracts*.

Needless to say, a number of challenges rest behind the scene of a design paradigm based on reusing off-the-shelf components: most notably, they arise from the mismatches between the behavior expected of the services to be included in a composition, and the behavior of the available services. At a very basic level, such mismatches may hinder the *compliance* [11] of the composite application, by leading it to a deadlock state, or trapping it into an infinite loop (a livelock). At a higher level, they may break the behavioral *conformance* with respect to the specification [1, 3, 4, 12].

There is a large body of work in the literature on how the mismatches may be attacked using techniques for component/service adaptation. In this paper, we develop a new theory of adaptable compositions that draws on the theory of contracts developed in [6], and supports the analysis of compliance and conformance of contract-based service compositions in a uniform, formally elegant setting. Our approach uses the filters introduced in [6] both as prescriptions of behavior (coercions to prevent service misbehavior) and as descriptors of the choreographic roles of the service components. We devise an algorithm to ensure the compliance of a composition by the automatic synthesis of an adapting filter. Further, we describe a technique, based on a simulation ordering, to verify the conformance of the adapted composition by contrasting the adapted contract against the associated *role filter*.

The paper is structured as follows. Sections 2 and 3 introduce our theory of service contract compositions and filters. Section 4 describes the algorithm for synthesizing the adapting filter. Section 5 shows how the theory supports the analysis of conformance. Section 6 discusses related work.

2. A Process Algebra for Service Composition

We represent service contracts as terms of a CCS-like [10] process calculus with recursion and operators for guarded and internal choice, but no parallel composition. Parallel composition arises in service contract compositions (SCC's for short), that we define after [3, 4, 11] as the parallel (and concurrent) composition of located contracts. We presuppose a denumerable set $\mathcal{N}$ of action names, ranged over by a, b, c, and a denumerable set Loc of location names, ranged over by l, m, n. The actions represent the basic units of observable behavior of the underlying services, while the location names specify the peers providing the services together with the ports on which the services are made available. The syntax of contract actions p, contracts

Table 1 Dynamics of Service Contracts and SCC's

Contract transitions: $\sigma \xrightarrow{\lambda} \sigma'$ with λ ranging over the following actions $\lambda ::= p \mid \tau$.

$$\mathbf{1}\checkmark \qquad \sum p_i.\sigma_i \xrightarrow{p_i} \sigma_i \qquad \sigma_1 \oplus \sigma_2 \xrightarrow{\tau} \sigma_i \quad (i=1,2)$$

$$\frac{\sigma\left\{{}^{\mathtt{rec}(X)\,\sigma}/_{X}\right\}\checkmark}{\mathtt{rec}(X)\,\sigma\checkmark} \qquad \frac{\sigma\left\{{}^{\mathtt{rec}(X)\,\sigma}/_{X}\right\}\xrightarrow{\lambda}\sigma'}{\mathtt{rec}(X)\,\sigma\xrightarrow{\lambda}\sigma'}$$

SCC transitions: $C \xrightarrow{\alpha}_L C'$ with α ranging over the actions $a_{n\to m} \mid \bar{a}_{n\to m}$ and $L \subset Loc$ is a finite set of locations.

$$\frac{\sigma\checkmark}{[\sigma]_n\checkmark} \qquad \frac{\sigma\xrightarrow{\tau}\sigma'}{[\sigma]_n\xrightarrow{\tau}_L[\sigma']_n} \qquad \frac{\sigma\xrightarrow{a}\sigma'}{[\sigma]_n\xrightarrow{a_{m\to n}}_L[\sigma']_n}\,(m\in L) \qquad \frac{\sigma\xrightarrow{\bar{a}@m}\sigma'}{[\sigma]_n\xrightarrow{\bar{a}_{n\to m}}_L[\sigma']_n}\,(m\neq n)$$

$$\frac{C_1\checkmark \quad C_2\checkmark}{C_1 \parallel C_2\checkmark} \qquad \frac{C_1\xrightarrow{a_{n\to m}}_L C_1' \quad C_2\xrightarrow{\bar{a}_{n\to m}}_L C_2'}{C_1\parallel C_2\xrightarrow{\tau}_L C_1'\parallel C_2'} \qquad \frac{C_1\xrightarrow{\alpha}_L C_1'}{C_1\parallel C_2\xrightarrow{\alpha}_L C_1'\parallel C_2} \qquad \frac{C_1\xrightarrow{\tau}_L C_1'}{C_1\parallel C_2\xrightarrow{\tau}_L C_1'\parallel C_2}$$

σ and compositions C is formally defined as follows:

p	$::=$	$\bar{a}@l \mid a$	send | receive a message
σ	$::=$	$\mathbf{1} \mid X$	termination | variable
	$\mid$	$\sum_{i=1}^{n} p_i.\sigma_i$	external choice
	$\mid$	$\sigma \oplus \sigma$	internal choice
	$\mid$	$\mathtt{rec}(X)\,\sigma$	recursion
C	$::=$	$[\sigma]_l$	located contract
	$\mid$	$C \parallel C$	composition

The contract $p.\sigma$ describes a service that waits for a message on p and then continues as σ; dually, $\bar{p}@l.\sigma$ sends a message on p to the service located at l and then behaves as σ. $\sum_{i=1}^{n} p_i.\sigma_i$ is a guarded, external choice: a service with this contract may choose any of the p_i and then continue as σ_i. We assume that $n \geq 0$ and $p_i \neq p_j$ whenever $i \neq j$, and note $\mathbf{0}$ a sum with $n = 0$. Internal choices may be unguarded: $\sigma \oplus \sigma'$ denotes a local choice between σ and σ'. The term $\mathtt{rec}(X)\,\sigma$ defines a possibly recursive contract: the recursion variable X may only occur guarded by a prefix in σ. Finally, $\mathbf{1}$ signals successful termination.

A composition must be well-formed [3, 4] to constitute a legal SCC, namely: (i) every component $[\sigma]_l$ must occur at a different location l of the composition, and (ii) the target of each output action at a given location must be different from the the location itself (i.e., the action $\bar{a}@l$ cannot occur inside a component $[\sigma]_l$). If $C = [\sigma_1]_{l_1} \parallel \cdots \parallel [\sigma_n]_{l_n}$ is a legal SCC, we say that C is an $\{l_1, \ldots, l_n\}$-SCC (dually, that $\{l_1, \ldots, l_n\}$ are the underlying locations for C).

Throughout, we assume that contracts are closed and that SCC's are well formed. Also, we often omit trailing $\mathbf{1}$'s.

Dynamics of contracts and SCCs. We define the dynamics of the calculus in terms of labelled transition semantics (and a success predicate), with rules reported in Table 1. We remark that the syntactic restriction on the external choice operator implies that for any given contract σ and action p, there is always at most one σ' such that $\sigma \xrightarrow{p} \sigma'$.

The transitions for SCC's are relative to the underlying set of locations. The transitions are mostly standard. Perhaps interestingly, the input transition is presented in an *early* style, by anticipating the source m of the signal on which the input action at n is going to synchronize. This gives us finer control on the synchronizations an input action may have in a given SCC. This, in turn, provides us with further control on how to shape an SCC by filtering.

Throughout, we omit the subscript L and write $\xrightarrow{\alpha}$ in place of $\xrightarrow{\alpha}_L$ when clear from the context. Also, we write $\Longrightarrow_L$ to note the reflexive and transitive closure of $\xrightarrow{\tau}_L$.

Finally, we introduce the notion of compliance [3]. Intuitively, a composition of services is compliant if it is deadlock and livelock free, i.e., it does not get stuck nor does it get trapped into infinite loops with no exit states.

Definition 2.1 [Compliance] Let C be a L-SCC. C is *compliant*, noted $C\downarrow$, if for every C' such that $C \Longrightarrow_L C'$ there exists C'' such that $C' \Longrightarrow_L C''$ and $C''\checkmark$.

Example 2.2 Table 2 shows an example of a compliant service contract composition. Three services are involved in this composition: C, S and B representing a *customer*, a *supplier* and a *bank*, respectively. The elementary actions represent business activities that result in messages being sent or received. For example, the action $\overline{\mathtt{Request}}@S$ undertaken by the customer results in a message being sent to

Table 2 A compliant SCC for E-payment: $C \parallel S \parallel B$

$$
\begin{array}{rcl}
C & = & [\overline{\texttt{Request}}@S.(\,(\overline{\texttt{PayDebit}}@S.\texttt{GetProd}.\mathbf{1} + \overline{\texttt{PayCredit}}@S.\texttt{GetProd}.\mathbf{1}) \oplus \\
 & & \qquad \overline{\texttt{PayCash}}@S.\overline{\texttt{GetCash}}@S.\texttt{GetProd}.\mathbf{1}\,)]_C \\
S & = & [\texttt{Request}.\overline{\texttt{Request}}@B.(\texttt{PayDebit}.\overline{\texttt{CheckDebit}}@B.\texttt{Done}.\overline{\texttt{GetProd}}@C.\mathbf{1} + \\
 & & \quad \texttt{PayCredit}.\overline{\texttt{CheckCredit}}@B.\texttt{Done}.\overline{\texttt{GetProd}}@C.\mathbf{1} + \\
 & & \quad \texttt{PayCash}.\texttt{GetCash}.\overline{\texttt{GetProd}}@C.\overline{\texttt{Done}}@B.\mathbf{1})]_S \\
B & = & [\texttt{Request}.(\texttt{CheckDebit}.\overline{\texttt{Done}}@S.\mathbf{1} + \texttt{CheckCredit}.\overline{\texttt{Done}}@S.\mathbf{1} + \texttt{Done}.\mathbf{1})]_B
\end{array}
$$

the supplier. In the example, the customer sends a request to the supplier and then decides whether to pay cash or by an electronic card, either a debit card or a credit card. The supplier contacts its referring bank and waits for the (internal) decision of the customer: in case of cash payment it accepts the money, then it ships the order and closes the communication with the bank, in case of electronic transaction it checks with the bank the availability of the money before shipping the order to the customer.

3. Filters and Component Adaptation

A filter is the specification of the legal flow of actions for an individual contract. We extend the definition of filters in [6] to allow recursive filters. The syntax is as follows:

$$f \in \mathcal{F} \;::=\; \mathbf{0} \mid \alpha.f \mid f \times f \mid f \otimes f \mid X \mid \texttt{rec}(X)\, f$$

Filters have a simple semantics, defined in Table 3. The transition relation is readily understood if we view a filter as a deterministic finite-state automaton accepting possibly infinite strings over the alphabet of SCC actions. If we take this view, $f_1 \times f_2$ and $f_1 \otimes f_2$ are directly realized as the union and the intersection of the automata f_1 and f_2.

Definition 3.1 [FILTER PRE-ORDER] The filter pre-order $f \overset{\alpha}{\leq} g$ is the largest relation such that if $f \overset{\alpha}{\longmapsto} f_\alpha$ then $g \overset{\alpha}{\longmapsto} g_\alpha$ and $f_\alpha \leq g_\alpha$.

We note $(\mathcal{F}, \sqsubseteq)$ the partial order induced by $\leq$: as usual we abuse the notation and identify a filter f with its equivalence class $[f]_\sim$, where $\sim$ is the symmetric closure of $\leq$. The union and intersection of filters represent the glb and lub operators for $(\mathcal{F}, \sqsubseteq)$. Futhermore, if we assume a finite alphabet A of actions, the set of filters $\mathcal{F}_A$ insisting on A forms a complete lattice with $\mathbf{0}$ as bottom and the identity filter $I_A \overset{def}{=} \texttt{rec}(X) \prod_{\alpha \in A} \alpha.X$ as top element.

Filters may be employed to block any contract transition that hinder the mutual compliance of the peers within an SCC: specifically, the application $f \triangleright [\sigma]_\ell$ blocks any action from $[\sigma]_\ell$ that is not explicitly enabled by f (cf. the second block of rules in Table 3).

Filters may be composed to help shape an SCC. Given a set of locations L, a composite L-filter F is finite map from the locations in L to filters: $\{\ell \to f_\ell \mid \ell \in L\}$. An L-filter may be applied to an L-SCC:

$$F \triangleright_L C ::= F[l_1] \triangleright [\sigma_1]_{l_1} \parallel \cdots \parallel F[l_n] \triangleright [\sigma_n]_{l_n}$$

When we write $F \triangleright_L C$ we tacitly assume that the underlying set of locations for both F and C is L. The operators of union and intersection, as well as the the ordering on filters extends directly to composite filters, as expected. Namely, for F and G L-filters and for $\bullet \in \{\times, \otimes\}$, we define:

$$
\begin{array}{rcl}
F \leq_L G & \text{iff} & F[\ell] \leq G[\ell] \quad \text{for all } \ell \in L \\
(F \bullet_L G)[\ell] & \overset{def}{=} & F[\ell] \bullet G[\ell] \quad \text{for all } \ell \in L
\end{array}
$$

We may then generalize the syntax of SCCs by allowing the term $F \triangleright_L C$ to account for the application of filters on the locations of the SCC. In the following, we will omit the subscript L when clear from the context. The dynamics of filtered SCCs derives directly by combining the transitions in the last block of rules in Table 1 with the ones for filtered locations in Table 3. Notice that when writing $F \triangleright C$ we may safely assume that C is not filtered, as nested filter applications like $F_1 \triangleright F_2 \triangleright C$ may equivalently be represented as the application of the intersection filter $(F_1 \otimes F_2) \triangleright C$.

Relevance and weak compliance. Being behavioral transformers, filters also help recover a notion of compliance for SCCs that are not compliant according to our current definition. Intuitively, a composition C is weakly compliant if it *can be made* compliant by filtering away all actions that may bring it to a deadlock or a livelock. We say that a composite filter F *fixes* C if $F \triangleright C$ is compliant.

Definition 3.2 [Weak Compliance] A composition C is *weakly compliant*, written $C \Downarrow$, if there exists a composite filter F that fixes C.

Example 3.3 Consider the SCC obtained from the one depicted in Table 2 by replacing the service B with $B^* = [\texttt{Request}.(\texttt{CheckCredit}.\overline{\texttt{Done}}@S.\mathbf{1} + \texttt{Done}.\mathbf{1})]_B$. In this case we lose compliance, as B^* only accepts credit cards: as

Table 3 Dynamics of Filtered SCC's

Transitions for filters

$$\alpha.f \stackrel{\alpha}{\longmapsto} f \qquad \frac{f\left\{{}^{\mathtt{rec}(X)\,f}/_{X}\right\} \stackrel{\alpha}{\longmapsto} f'}{\mathtt{rec}(X)\,f \stackrel{\alpha}{\longmapsto} f'} \qquad \frac{f \stackrel{\alpha}{\longmapsto} f_\alpha \qquad g \stackrel{\alpha}{\longmapsto} g_\alpha}{f \otimes g \stackrel{\alpha}{\longmapsto} f_\alpha \otimes g_\alpha}$$

$$\frac{f \stackrel{\alpha}{\longmapsto} f_\alpha \qquad g \stackrel{\alpha}{\longmapsto} g_\alpha}{f \times g \stackrel{\alpha}{\longmapsto} f_\alpha \times g_\alpha} \qquad \frac{f \stackrel{\alpha}{\longmapsto} f_\alpha \qquad g \not\stackrel{\alpha}{\longmapsto}}{f \times g \stackrel{\alpha}{\longmapsto} f_\alpha} \qquad \frac{f \not\stackrel{\alpha}{\longmapsto} \qquad g \stackrel{\alpha}{\longmapsto} g_\alpha}{f \times g \stackrel{\alpha}{\longmapsto} g_\alpha}$$

Transitions for filtered locations

$$\frac{[\sigma]_n \stackrel{\alpha}{\longrightarrow} [\sigma']_n \qquad f \stackrel{\alpha}{\longmapsto} f'}{f \triangleright [\sigma]_n \stackrel{\alpha}{\longrightarrow} f' \triangleright [\sigma']_n} \qquad \frac{[\sigma]_n \stackrel{\tau}{\longrightarrow} [\sigma']_n}{f \triangleright [\sigma]_n \stackrel{\tau}{\longrightarrow} f \triangleright [\sigma']_n} \qquad \frac{[\sigma]_n \checkmark}{f \triangleright [\sigma]_n \checkmark}$$

a result the composition gets stuck in case the customer and the supplier agree on a payment by debit card. However the composition can be fixed by filtering out the actions leading the customer to pay by debit card.

In the next section we present an algorithm that given a composition C infers a composite filter F that fixes C, whenever such F exists. The algorithm is so structured as to guarantee two important properties on the inferred filter. On the one hand, the filter is as permissive as possible, in that it is the greatest (with respect to the pre-order $\leq$) among the filters that fix C. On the other side, the inferred filter is *relevant*, i.e., minimal in size: for any computation state reached by the SCC via a series of τ transitions (local moves or synchronizations), the filter only enables actions that may be attempted at that state (either directly, or via a local choice), by one of the components of the SCC.

While the notion of relevance is intuitive, the presence of the local moves makes its definition somewhat involved. We introduce some notation to help (i) to distinguish a local move from a synchronization, and (ii) to identify the contribution of every location in a synchronization. We write

- $C \stackrel{\tau}{\hookrightarrow} C'$ iff $C \stackrel{\tau}{\longrightarrow} C'$ because $C \equiv [\sigma]_\ell \parallel C''$, $C' \equiv [\sigma']_\ell \parallel C''$ and $[\sigma]_\ell \stackrel{\tau}{\longrightarrow} [\sigma']_\ell$.
- $C \stackrel{\{a_{n\to m}\}}{\hookrightarrow} C'$ iff $C \stackrel{\tau}{\longrightarrow} C'$ because $C \equiv C_1 \parallel C_2$, $C_1 \stackrel{a_{n\to m}}{\longrightarrow} C_1'$, $C_2 \stackrel{\bar{a}_{n\to m}}{\longrightarrow} C_2'$, and $C' \equiv C_1' \parallel C_2'$.

We let φ range over the labels $\{a_{n\to m}\}$ and τ. We define $\stackrel{\tau}{\hookrightarrow\!\!\!\!\!\rightarrow} \stackrel{def}{=} \stackrel{\tau}{\hookrightarrow} \cdots \stackrel{\tau}{\hookrightarrow}$ and $\stackrel{\{a_{n\to m}\}}{\hookrightarrow\!\!\!\!\!\rightarrow} \stackrel{def}{=} \stackrel{\tau}{\hookrightarrow\!\!\!\!\!\rightarrow} . \stackrel{\{a_{n\to m}\}}{\hookrightarrow} . \stackrel{\tau}{\hookrightarrow\!\!\!\!\!\rightarrow}$.

Definition 3.4 [RELEVANCE] Let $\mathcal{C}$ be a non-empty set of L-SCCs. A filter f is *ℓ-relevant* in $\mathcal{C}$, written $f \propto_\ell \mathcal{C}$, if whenever $f \stackrel{\alpha}{\longmapsto} \widehat{f}$ one has $\alpha \in \{a_{\_\to\ell}, \bar{a}_{\ell\to\_}\}$ and there exists $\widehat{C} \in \mathcal{C}$ such that[1] $\widehat{C} \stackrel{\{\alpha\}}{\hookrightarrow\!\!\!\!\!\rightarrow}$ and $\widehat{f} \propto_\ell \{C' \mid \widehat{C} \stackrel{\{\alpha\}}{\hookrightarrow\!\!\!\!\!\rightarrow} C'\}$.

[1]This notation is loose: when $\alpha = \bar{a}_{\ell\to\_}$, $\{\alpha\}$ is, in fact, $\{a_{\ell\to\_}\}$

A composite L-filter F is *relevant* for $\mathcal{C}$, written $F \propto \mathcal{C}$ iff $F[\ell] \propto_\ell \mathcal{C}$ for all $\ell \in L$. Finally, a composite L-filter is relevant for an L-SCC C if $F \propto \{C\}$.

4. Synthesis of the Maximal Relevant Filter

We now describe the algorithm that, given an SCC, synthesizes the $\sqsubseteq$-greatest relevant filter that fixes the SCC.

Given a composition C, the algorithm keeps track of the reachable states of the computation in C so as to filter out the actions that may bring C to a deadlock or a livelock. Since C is finite state, the reachable states may be organized into a finite state reduction graph.

A reduction graph is a directed graph $G = (V, E)$ with labeled edges and vertices. The vertices in V represent the reachable states of the underlying composition C. With each $\mathbf{v} \in V$ we associate two fields: $state[\mathbf{v}]$ gives the computation state (i.e., the derivative C' of the initial state C) associated with $\mathbf{v}$; $result[\mathbf{v}]$ is a tag, SUCC, FAIL or UNDEF, that informs on the possible outcomes of the computation starting off at this state. An edge in E is a triple $(\mathbf{u}, \mathbf{v})_\varphi$ representing the transition $state[\mathbf{u}] \stackrel{\varphi}{\longrightarrow} state[\mathbf{v}]$. The reduction graph only traces the states reached by means of synchronizations or internal moves, thus disregarding all states reached by labeled transitions.

Reduction graphs are stored in adjacency list representation, so that the set of outgoing edges for each $\mathbf{u} \in V$ can be retrieved as $Adj[\mathbf{u}]$: thus $(\mathbf{u}, \mathbf{v})_\varphi \in E$ iff $(\varphi, \mathbf{v}) \in Adj[\mathbf{u}]$. We also write $Adj[\mathbf{u}, \varphi]$ for the set $\{\mathbf{v} \in V \mid (\mathbf{u}, \mathbf{v})_\varphi \in E\}$. Vertices with no outgoing edges are called leaves. We denote by $root[G]$ the vertex representing the initial state C.

The algorithm involves several steps. The first step builds the reduction graph for the given SCC. The second step propagates the *result* labels to verify whether the SCC admits a filter, i.e., whether its reduction graph contains at least one successful path from the root to a final state. The

third step extracts the sub-graph of the successful paths in the reduction graph. The algorithm fails at this stage if the extracted sub-graph is empty. Otherwise the algorithm succeeds by synthesizing the filter from the success sub-graph.

Building the reduction graph. The reduction graph $G = (V, E)$ is built in a top-down manner from a given initial SCC term C through the function `BuildGraph`(C). The construction iteratively explores the set of states reached from C. The auxiliary function `NewVertex`(C) creates a new vertex with UNDEF result label and state C. The loop terminates when all the vertices reached have been visited.

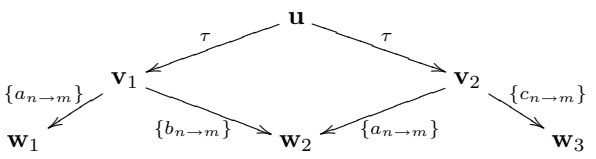
Function `BuildGraph`(C)

Input: A choreography C
Output: $G = (V, E)$ the reduction graph of C

$\mathbf{r} :=$ `newVertex`(C); $V := \{\mathbf{r}\}$; $E := \emptyset$; $W := \emptyset$;
while $(V \setminus W \neq \emptyset)$ **do**
  $\mathbf{u} :=$ `select`$(V \setminus W)$;
  if $state[\mathbf{u}]$ ✓ **then**
    $result[\mathbf{u}] :=$ SUCC;
  else if $state[\mathbf{u}] \overset{\varphi}{\not\longrightarrow}$ **then**
    $result[\mathbf{u}] :=$ FAIL;
  foreach $\widehat{C} : state[\mathbf{u}] \overset{\varphi}{\longrightarrow} \widehat{C}$ **do**
    $\mathbf{v} :=$ `select`$(\{\mathbf{w} \in V \mid state[\mathbf{w}] = \widehat{C})$;
    if $\mathbf{v}$ = `NULL` **then**
      $\mathbf{v} :=$ `newVertex`$(\widehat{C})$; $V := V \cup \{\mathbf{v}\}$;
    $E := E \cup \{(\mathbf{u}, \mathbf{v})_\varphi\}$;
  $W := W \cup \{\mathbf{u}\}$;
$G := (V, E)$; $root[G] := \mathbf{r}$;
return G;

Lemma 4.1 If G is a graph generated by the function `BuildGraph`(C), then all the internal vertices of G are labeled as UNDEF. Only the leaf with state $[\mathbf{1}]_{\ell_1} \parallel \ldots \parallel [\mathbf{1}]_{\ell_k}$, if it exists, is labeled as SUCC and it is denoted $succ[G]$. All the other leafs are labeled as FAIL.

Example 4.2 The graph G below is generated by the configuration $C_1 \parallel C_2$, with $C_1 = [(a.d.\mathbf{1} + b.\mathbf{1}) \oplus (a.\mathbf{1} + c.e.\mathbf{1})]_m$ and $C_2 = [\bar{a}@m.\mathbf{1} + \bar{b}@m.\mathbf{1} + \bar{c}@m.\mathbf{1}]_n$, and labels: $result[\mathbf{w}_1] = result[\mathbf{w}_3] =$ FAIL, $result[\mathbf{u}] = result[\mathbf{v}_1] = result[\mathbf{v}_2] =$ UNDEF, $result[\mathbf{w}_2] =$ SUCC, and $succ[G] = \mathbf{w}_2$.



Labelling the reduction graph. Let us first introduce some auxiliary definitions. Let $locs(\varphi)$ be $\{m, n\}$ in case $\varphi = \{a_{m\to n}\}$, and $\emptyset$ in case $\varphi = \tau$. Then, let $G = (V, E)$ be a reduction graph, and $\varphi = \{a_{m\to n}\}$.

- A path $\pi = (\mathbf{u}, \mathbf{u}_1)_{\varphi_1}, \ldots, (\mathbf{u}_{n-1}, \mathbf{v})_{\varphi_n}$ from $\mathbf{u}$ to $\mathbf{v}$ in G is *φ-free* if $locs(\varphi) \cap locs(\varphi_i) = \emptyset$ for all i's.

- A vertex $\mathbf{v}$ is a *φ-free descendant* of $\mathbf{u}$ in G (dually, $\mathbf{u}$ is a *φ-free ancestor* of $\mathbf{v}$) if there is a φ-free path from $\mathbf{u}$ to $\mathbf{v}$.

- A vertex $\mathbf{u}$ *yields a conflict on* φ if $\mathbf{u}$ has two distinct φ-free descendants $\mathbf{v}_1$ and $\mathbf{v}_2$ such that $(\mathbf{v}_1, \mathbf{w}_1)_\varphi$ and $(\mathbf{v}_2, \mathbf{w}_2)_\varphi \in E$ and $result[\mathbf{w}_1] \neq result[\mathbf{w}_2] \neq$ UNDEF.

- A vertex $\mathbf{v}$ has a conflict on φ in G, noted $Conflict_G(\varphi, \mathbf{v})$ if $\mathbf{v}$ has a φ-free ancestor yielding a conflict on φ.

Intuitively, a graph G represents a compliant SCC if every path in G starting form $root[G]$ can be extended to reach $succ[G]$. To ensure compliance, filters must then prune the graph by banning all the 'bad' synchronizations that lead to a node that cannot reach $succ[G]$, and by preserving all the 'good' synchronizations that converge to $succ[G]$. Due to the presence of internal choices, the same synchronization can look good at one point, but actually be bad. The definition of conflict captures formally this notion of ambiguous synchronizations.

Example 4.3 In the graph G of Example 4.2 it holds that $Conflict_G(\{a_{n\to m}\}, \mathbf{u})$. Indeed, focus on $\mathbf{v}_2$. In order to guarantee compliance, we have to ban $(\mathbf{v}_2, \mathbf{w}_3)_{\{c_{n\to m}\}}$ and to preserve $(\mathbf{v}_2, \mathbf{w}_2)_{\{a_{n\to m}\}}$. On the other hand, at $\mathbf{v}_1$ we have to prune the edge $(\mathbf{v}_1, \mathbf{w}_1)_{\{a_{n\to m}\}}$ as it leads to a failure. Since $\mathbf{u}$ reaches $\mathbf{v_1}$ and $\mathbf{v_2}$ via τ actions, and a filter has no control on τ's, the candidate filter should allow and, at the same time, prohibit the action $a_{n\to m}$ at location m and $\bar{a}_{m\to n}$ at location n. This is clearly impossible, hence the choreography $C_1 \parallel C_2$ cannot be fixed.

The next step consists in *labelling* the graph to propagate the result label from $succ[G]$ back towards $root[G]$. This is achieved by the procedure `LabelGraph`(G) which runs the auxiliary procedure `PushLabels`(G) twice. The latter receives a graph G generated by `BuildGraph`(C) and updates just the UNDEF vertices of G. The result label at each vertex $\mathbf{u}$ is set to FAIL if there exists at least one silent transition from $\mathbf{u}$ to a FAIL vertex; it is set to SUCC if either there are no silent transitions from $\mathbf{u}$ to a FAIL vertex and there exists a silent transition from $\mathbf{u}$ to a SUCC vertex or there exists one non-silent and non-conflicting transition from $\mathbf{u}$ to a SUCC vertex. The procedure iteratively examines all the vertices in the graph until it reaches a fixed point.

Lemma 4.4 The following conditions are invariant for the main loop in `PushLabels`(G). For every node $\mathbf{u}$ in G: (i) $result[\mathbf{u}]$ changes to FAIL and SUCC at least once; (ii)

Procedure `PushLabels(`G`)`

Input: A reduction graph $G = (V, E)$
Output: The graph G updated

```
done := false;
while ¬ done do
    done := true;
    foreach u ∈ V do
        succ := false;  fail := false;
        if Adj[u, τ] ≠ ∅ then
            if ∃v ∈ Adj[u, τ] : result[v] = FAIL then
                fail := true;
            else if ∃v∈Adj[u, τ] : result[v] = SUCC
            then
                succ := true;
        else if ∃(φ, v) ∈ Adj[u] ∧ result[v] =
        SUCC ∧ ¬Conflict(φ, u) then
            succ := true;
        if succ ∧ result[u] ≠ SUCC then
            result[u] := SUCC;  done := false;
        else if fail ∧ result[u] ≠ FAIL then
            result[u] := FAIL;  done := false
```

$result[\mathbf{u}]$ never changes to UNDEF; (iii) when $result[\mathbf{u}] =$ FAIL, then the label does not change anymore during the computation.

Moreover, at the end of the loop the variable *done* is `false` iff some node $\mathbf{u}$ has changed its status during the current loop. Hence the procedure `PushLabels(`G`)` terminates.

Procedure `LabelGraph(`G`)`

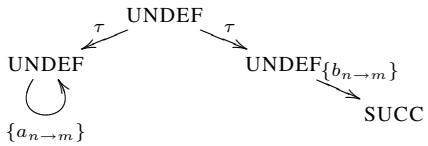
Input: A reduction graph $G = (V, E)$
Output: The graph G updated

```
PushLabels(G);
foreach u ∈ V do
    if result[u] = UNDEF then
        result[u] := FAIL;
PushLabels(G);
```

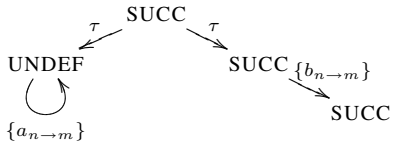
Lemma 4.5 After the call to `PushLabels(`G`)`, the following conditions hold for every node $\mathbf{u}$ in G: (i) $result[\mathbf{u}] =$ FAIL iff either there exists no $(\mathbf{u}, \mathbf{v})_\varphi \in E$ such that $result[\mathbf{v}] =$ SUCC and $\neg Conflict_G(\varphi, \mathbf{u})$ or there exists $(\mathbf{u}, \mathbf{v})_\tau \in E$ such that $result[\mathbf{v}] =$ FAIL; (ii) $result[\mathbf{u}] =$ SUCC iff there exists no $(\mathbf{u}, \mathbf{v})_\tau \in E$ such that $result[\mathbf{v}] =$ FAIL and there exists either $(\mathbf{u}, \mathbf{v})_\tau \in E$ such that $result[\mathbf{v}] =$ SUCC or $(\mathbf{u}, \mathbf{v})_\varphi \in E$ with $\varphi \neq \tau$, $\neg Conflict_G(\varphi, \mathbf{u})$ and $result[\mathbf{v}] =$ SUCC; (iii) $result[\mathbf{u}] =$ UNDEF iff $result[\mathbf{v}] =$ UNDEF for every node $\mathbf{v}$ that is reachable form $\mathbf{u}$.

By Lemmata 4.5 and 4.1, it follows that if $result[\mathbf{u}] =$ UNDEF after a run of `PushLabels(`G`)` then it is impossible to reach any leaf of G from $\mathbf{u}$. Hence $\mathbf{u}$ is inside a cycle of UNDEF vertices. In this case, vertex $\mathbf{u}$ must be marked FAIL. This is accomplished by the `LabelGraph(`G`)` procedure which first runs `PushLabels(`G`)`, then sets to FAIL every UNDEF vertex, and finally it runs `PushLabels(`G`)` again.

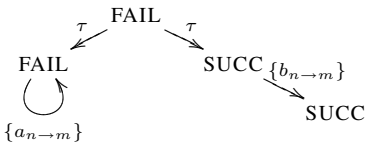
Example 4.6 The reduction graph G below is generated by $[(\mathtt{rec}(X)\, a.X) \oplus b.\mathbf{1}]_m \parallel [(\mathtt{rec}(X)\, \bar{a}@m.X) + \bar{b}@m.\mathbf{1}]_n$



After the first run of `PushLabels(`G`)`, we obtain:



Now the UNDEF node must clearly be set to FAIL. Thus, a further run of `PushLabels(`G`)` is needed to propagate the failure to the root.



The following post-condition for `LabelGraph(`G`)` holds. We say that a path π in G is *successful* if $result[\mathbf{u}] =$ SUCC for every node $\mathbf{u}$ in π, otherwise π is *unsuccessful*. A node $\mathbf{u}$ is *root-successful* if it is reachable from *root*$[G]$ via a successful path, otherwise it is *root-unsuccessful*.

Lemma 4.7 Let G be generated by `BuildGraph(`C`)`, then after the call `LabelGraph(`G`)` it holds: (i) for every root-successful node $\mathbf{u}$ in G, there exists a successful path from $\mathbf{u}$ to *succ*$[G]$; (ii) for every root-unsuccessful node $\mathbf{u}$ in G, either there exists no path from $\mathbf{u}$ to *succ*$[G]$ or there exist a path π from $\mathbf{u}$ to $\mathbf{v}$ and $\mathbf{u}', \mathbf{v}'$ such that $result[\mathbf{v}]$=FAIL and $(\mathbf{u}', \mathbf{v}')_\tau \in \pi$.

Extracting the success subgraph. Item (i) of Lemma 4.7 hints that the root-successful vertices of G represent the computation of a compliant configuration, as they satisfy the requirements of Definition 2.1. Item (ii) of the same lemma hints that if we consider just a root-unsuccessful node, we make this ideal computation break compliance.

The next step of the algorithm computes the sub-graph of G that only includes the root-successful vertices, along with the edges in the successful paths. This computation is accomplished by the $\texttt{SuccessGraph}(G)$ function below.

Function $\texttt{SuccessGraph}(G)$

Input: A closed reduction graph $G = (V, E)$
Output: $G' = (V', E')$ the success sub-graph of G

$V' := (\mathit{result}[\mathit{root}[G]] = \text{SUCC})\,?\,\{\mathit{root}[G]\} : \emptyset$;
$E' := \emptyset$; $\mathit{done} := \texttt{false}$;
while $\neg\,\mathit{done}$ **do**
  $\mathit{done} := \texttt{true}$;
  foreach $(\mathbf{u}, \mathbf{v})_\varphi \in E \setminus E'$ **do**
    if $\mathbf{u} \in V' \wedge \mathit{result}[\mathbf{v}] = \text{SUCC} \wedge \neg \mathit{Conflict}(\varphi, \mathbf{u})$ **then**
      $V' := V' \cup \{\mathbf{v}\}$; $E' := E' \cup \{(\mathbf{u}, \mathbf{v})_\varphi\}$;
      $\mathit{done} := \texttt{false}$
return $G' = (V', E')$;

Lemma 4.8 Let $G = (E, V)$ be a graph generated by the procedure $\texttt{BuildGraph}(C)$, and let $G' = (E', V')$ be the graph generated by $\texttt{SuccessGraph}(G)$. Then $\mathbf{u} \in V'$ if and only if $\mathbf{u}$ is root-successful in G.

Synthesizing the filter. The filter is extracted from the success graph by projecting the complementary actions of each synchronization step to the contributing locations. Let $G = \texttt{BuildGraph}(C)$, $G' = \texttt{SuccessGraph}(G)$, and $F_C^{Alg} = \texttt{ExtractFilter}_L(\mathit{root}[G], \emptyset, G')$. Thanks to item (i) of Lemma 4.7 we derive the correctness of the algorithm. By construction, F_C^{Alg} is relevant for C. Furthermore, item (ii) of Lemma 4.7 and Lemma 4.8 imply that F_C^{Alg} is maximum among the relevant filters that fix C.

Theorem 4.9 [SOUNDNESS AND MAXIMALITY] Let C be a SCC. Then $F_C^{Alg} \triangleright C$ is compliant. Moreover, if a filter F fixes C and is relevant for C, then $F \leq F_C^{Alg}$.

Corollary 4.10 [COMPLETENESS] Let C be a service-contract composition. If C is weakly compliant, then the algorithm succeeds and extracts the maximum relevant filter.

Example 4.11 Consider the service contract composition $C \parallel S \parallel B^*$ of Example 3.3. A run of our algorithm will produce the filter F depicted in Table 4.

5. Conformance Validation

Filters have an additional purpose in our theory, as behavioral descriptors: in fact, we employ them to specify the intended roles of a choreography. This is possible as filters are built around actions that provide accurate information on the end-points involved in the operations of message sent/reception. This is precisely what we need when projecting a choreographic design on the components to define the role (i.e., the expected behavior) of each component.

Function $\texttt{ExtractFilter}_L(\mathbf{u}, U, G)$

Input: $G = (V, E)$ a success graph. $\mathbf{u} \in V, U \subseteq V$
Output: F, an L-composite filter

$F[\ell] := \mathbf{0}$ for all $\ell \in L$.;
if $\mathit{state}[\mathbf{u}]$ ✓ **then**
  return F;
if $\mathbf{u} \in U$ **then**
  $\mathit{rec}[\mathbf{u}] := \texttt{true}$; **return** $(X_\mathbf{u}, \ldots, X_\mathbf{u})$;
foreach $(\varphi, \mathbf{v}) \in \mathit{Adj}[\mathbf{u}]$ **do**
  $F_\mathbf{v} := \texttt{ExtractFilter}_L(\mathbf{v}, U \cup \{\mathbf{u}\}, G)$;
  foreach location $\ell \in L$ **do**
    if $\varphi = \{a_{\ell \to \_}\}$ **then**
      $F[\ell] := F[\ell] \times \bar{a}_{\ell \to \_}.F_\mathbf{v}[\ell]$;
    else if $\varphi = \{a_{\_ \to \ell}\}$ **then**
      $F[\ell] := F[\ell] \times a_{\_ \to \ell}.F_\mathbf{v}[\ell]$;
    else
      $F[\ell] := F[\ell] \times F_\mathbf{v}[\ell]j$;
if $\mathit{rec}[\mathbf{u}] = \texttt{true}$ **then**
  foreach $\ell \in L :\ X_\mathbf{u} \in \mathit{fv}(F[\ell])$ **do**
    $F[\ell] := \texttt{rec}(X_\mathbf{u})\,F[\ell]$;
return F;

We then check the conformance of a composition against a choreographic specification as we outline next. We assume that choreography specifications are given directly in projected form, as a set $\Phi = \{\Phi(\ell) \mid \ell \in L\}$ of role specifications, with each role specified by a filter. We let $\stackrel{\alpha}{\Longrightarrow}$ denote the weak transition $\stackrel{\tau}{\longrightarrow}^* \stackrel{\alpha}{\longrightarrow} \stackrel{\tau}{\longrightarrow}^*$, and $\stackrel{\alpha_1 \ldots \alpha_n}{\Longrightarrow}$ a sequence of such transitions (it stands for $\stackrel{\tau}{\longrightarrow}^*$ when $n = 0$).

Definition 5.1 [ROLE SUPPORT] A binary relation $\mathcal{S}$ between (role) filters and service contracts is a role simulation if whenever $(\phi, [\sigma]_\ell) \in \mathcal{S}$ one has:

- if $\phi \stackrel{\alpha}{\longmapsto} \phi'$ there exist $\alpha_1, \ldots, \alpha_n, \sigma'$ such that $[\sigma]_\ell \stackrel{\alpha_1 \ldots \alpha_n}{\Longrightarrow} \cdot \stackrel{\alpha}{\longrightarrow} [\sigma']_\ell$ with $\alpha \neq \alpha_i$ and $(\phi', [\sigma']_\ell) \in \mathcal{S}$;
- if $[\sigma]_\ell \stackrel{\alpha}{\Longrightarrow} [\sigma']_\ell$ then either $\phi \stackrel{\alpha}{\longmapsto} \phi'$ and $(\phi', [\sigma']_\ell) \in \mathcal{S}$ or $\phi \stackrel{\alpha}{\not\longmapsto}$ and $(\phi, [\sigma']_\ell) \in \mathcal{S}$.

A contract $[\sigma]_\ell$ supports a role ϕ, noted $\phi \lhd [\sigma]_\ell$, if $(\phi, [\sigma]_\ell) \in \mathcal{S}$ with $\mathcal{S}$ role simulation. A L-SCC C supports a specification Φ, written $\Phi \lhd_L C$ iff $\Phi(\ell) \lhd [\sigma_\ell]_\ell$ for all $\ell \in L$.

A role simulation allows the simulating contract to have additional observable behavior over the simulated role, but only if it preserves the branching structure of the role.

Table 4 The Filter F for $C \parallel S \parallel B^*$

$$
\begin{array}{lcl}
F[C] & = & \overline{\texttt{Request}}_{C\to S}.(\,\overline{\texttt{PayCredit}}_{C\to S}.\texttt{GetProd}_{S\to C}.\mathbf{0} \times \overline{\texttt{PayCash}}_{C\to S}.\overline{\texttt{GetCash}}_{C\to S}.\texttt{GetProd}_{S\to C}.\mathbf{0}\,) \\
F[S] & = & \texttt{Request}_{C\to S}.\overline{\texttt{Request}}_{S\to B}.(\,\texttt{PayCredit}_{C\to S}.\overline{\texttt{CheckCredit}}_{S\to B}.\texttt{Done}_{B\to S}.\overline{\texttt{GetProd}}_{S\to C}.\mathbf{0} \times \\
 & & \qquad\qquad \texttt{PayCash}_{C\to S}.\texttt{GetCash}_{C\to S}.\overline{\texttt{GetProd}}_{S\to C}.\overline{\texttt{Done}}_{S\to B}.\mathbf{0}\,) \\
F[B^*] & = & \texttt{Request}_{S\to B}.(\texttt{CheckCredit}_{S\to B}.\overline{\texttt{Done}}_{B\to S}.\mathbf{0} \times \texttt{Done}_{S\to B}.\mathbf{0})
\end{array}
$$

Definition 5.2 [CONFORMANCE] A contract composition C is Φ-conformant iff F_C^{Alg} exists and $\Phi \lhd (F_C^{Alg} \rhd C)$.

Requiring $F_C^{Alg} \rhd C$ to support Φ gives a guarantee of functional completeness for the filtered choreography. On the other hand, the properties of F_C^{Alg} ensure that the additional behavior exposed by $(F_C^{Alg} \rhd C)$, if any, is safe. To illustrate, let $\phi = a_{m\to\ell}.\phi'$ be the role, and $\sigma = a.\sigma_1 \oplus b.a.\sigma_2$ be the contract at ℓ. Then, $\phi \lhd [\sigma]_\ell$ whenever $\phi' \lhd [\sigma_1]_\ell$ and $\phi' \lhd [\sigma_2]_\ell$. Indeed, when $[\sigma]_\ell$ is a filtered location, we know that the 'spurious' action b does not deadlock (otherwise $b.a.\sigma_2$ would have been filtered away). On the other hand $\phi \not\lhd [a.\sigma_1 \oplus b]_\ell$ as the service at ℓ may deliberately choose not to execute a as required by its role.

6. Related Work

There is a number of proposals in the literature dealing with composition, interoperation and adaption within SOA. Closer to our work are the proposals that address the compliance problem within choreographies [3, 4, 11]. In [11] the problem is reduced to a bipartite compatibility problem: each partner can check locally its compliance by verifying its compatibility with the aggregation of all its partners. Our present concern is more general, as we also devise a technique for adapting the aggregation to solve the behavioral mismatches that hinder compliance.

Component adaptation has itself received much interest in the literature. Among the many approaches [2, 5, 7], the closest to ours is perhaps [8], where the authors develop a method for the automated extraction of an adapter for a service composition out of the LTS. The adapter is deployed as an independent component that orchestrates the execution of its peers to ensure safety of the execution flow. More recently, in [9], the approach has been refined with a technique for projecting the global adapter onto the individual system components. While we share some of the initial ideas with these two papers, specifically the idea of extracting the adapter from the LTS, our approach is different for a number of design choices and technical issues. First, we extract the filters directly from the individual components rather than obtaining them by a projection from a global adapter. Secondly, our filter fully preserves the action sequence in the original components, whereas the adapter synthesized in [9] may require a reorder. Also, our formalization of adapters as filters makes it possible to formulate, and prove formally, a precise characterization of the properties satisfied by the extracted filter (i.e., relevance and maximality). Finally, our theory provides a uniform setting for a combined analysis of compliance and conformance in terms of well-established behavioral techniques.

References

[1] M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella. Verifying the conformance of web services to global interaction protocols: A first step. In *WS-FM'05*, volume 3670 of *LNCS*, pages 257–271. Springer, 2005.

[2] A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *Journal of Systems and Software*, 74(1):45–54, 2005.

[3] M. Bravetti and G. Zavattaro. Towards a unifying theory for choreography conformance and contract compliance. In *SC'07*, volume 4829 of *LNCS*, pages 34–50. Springer, 2007.

[4] M. Bravetti and G. Zavattaro. Contract compliance and choreography conformance in the presence of message queues. In *WS-FM'08*, LNCS. Springer, 2008. To appear.

[5] A. Brogi, C. Canal, and E. Pimentel. Component adaptation through flexible subservicing. *Science of Computer Programming*, 63(1):39–56, 2006.

[6] G. Castagna, N. Gesbert, and L. Padovani. A theory of contracts for web services. In *POPL'08*, pages 261–272. ACM press, 2008.

[7] P. Inverardi and M. Tivoli. Deadlock free software architectures for com/dcom applications. *Journal of Systems and Software*, 65(3):173–183, 2005.

[8] R. Mateescu, P. Poizat, and G. Salaün. Behavioral adaptation of component compositions based on process algebra encodings. In *ASE'07*, pages 385–388. ACM Press, 2007.

[9] T. Melliti, P. Poizat, and S. Ben Mokhtar. Distributed behavioural adaptation for the automatic composition of semantic services. In *FASE'08*, volume 4961 of *LNCS*, pages 146–162, 2008.

[10] R. Milner. *Communication and Concurrency*, volume 92 of *Prentice Hall International Series in Computer Science*. Prentice Hall, 1989.

[11] M. Tarek, C. Boutrous-Saab, and S. Rampacek. Verifying correctness of web services choreography. In *ECOWS'06*, pages 306–318, 2006.

[12] W. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, and H. Verbeek. Conformance checking of service behavior. *ACM Transactions on Internet Technology, Special Issue on Middleware for Service-Oriented Architectures*, 2008. To appear.